НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

«До захисту допущено»
В.о. завідувача кафедри

_____ М.В.Грайворонський
(підпис)
“___”_____2019 р.

Дипломна робота

на здобуття ступеня бакалавра

з напряму підготовки 6.170101 «Безпека інформаційних і комунікаційних систем»

на тему: Підвищення кваліфікації тестувальника XSS/CSRF уразливостей із використанням автоматизованого перетворювача коду

Виконав (-ла): студент (-ка) _4_ курсу, групи ___ФБ-51_____
(шифр групи)

_____Оніщенко Антон_____
(прізвище, ім’я, по батькові) (підпис)

Керівник: доцент к.т.н. Литвинова Т.В..
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____ _____ _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ - 2019 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)

Напрям підготовки 6.170101 «Безпека інформаційних і комунікаційних систем»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ М.В.Грайворонський
(підпис)
«___»_____2019 р.

ЗАВДАННЯ
на дипломну роботу студенту

_____Оніщенко Антон_____
(прізвище, ім'я, по батькові)

1. Тема роботи Підвищення кваліфікації тестувальника XSS/CSRF уразливостей із використанням автоматизованого перетворювача коду , науковий керівник роботи доцент к.т.н. Литвинова Т.В. _____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «____ » 2019 р. № _____

2. Термін подання студентом роботи 10 червня 2019 р.

3. Вихідні дані до роботи _____
_____
_____

4. Зміст роботи_____
_____
_____
_____
_____

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо) _____
_____
_____
_____

6. Дата видачі завдання _____

РЕФЕРАТ

Робота обсягом 72 сторінки містить 24 ілюстрацій, 8 таблиць та 20 літературних посилань.

Метою і завданням дослідження дипломної роботи є захищення веб-застосунків на основі створення інструменту, що може вводити варіації вразливостей у існуючий сайт чи застосунок. Цей інструмент повинен мати можливість вбудовувати уразливості, які складно виявити сканерами уразливостей (що розміщені у вільному доступі).

Об’єктом дослідження є уразливості, сканери уразливостей та інформаційні процеси при потоці даних та їх обробка для знаходження не захищених місць у коді програм.

В якості предмету дослідження розглядаються захищеність php коду у застосунках та веб-сайтах. Результати роботи викладені у вигляді таблиці та методу.

Результати роботи можуть бути використані для підвиення результативності роботи QA тестувальників. Дана робота може бути теоретично посібником для пен-тестерів і бути включеною у програму навчання на реальних вразливих прикладах веб-застосунків із вразливостями і допомагали їм у їх професійному навчанні.

ВЕБ-ЗАСТОСУНОК, JAVASCRIPT, PHP, XSS/CSRF УРАЗЛИВОСТІ, БЕЗПЕКА ІНФОРМАЦІЙНО І КОМУНІКАЦІЙНИХ СИСТЕМ

РЕФЕРАТ

Робота обьемом 72 страниц содержит 24 иллюстраций, 8 таблиц и 20 литературных ссылок.

Цулью и заданием исследования данной дипломной работы есть защита веб-приложений на основе создания инструмента-програмки, что может создавать и вводить вариации уязвимостей у существуйющий сайт или приложение. Этот инструмент доджен иметь возможность встраивать уязвимости, что сложо найти сканером уязвимостей (что можно найти в свободном доступе).

Обьект исследования – обьектом исследования есть уязвимости, сканеры уязвимостей и информационные потоки данных, их обработка для нахождения не защищенных мест.

В качестве предмета исследования рассматриваеться защищенность php кода в приложениях та веб-сайтах.

Результаты выложены в качестве таблиц и метода, что демонстрирует защищенность выбраных для анализа приложений.

Результаты работы могут быть использованы для повышения результативности QA тестировщиков. Данная работа может быть теореическим пособием для пен-тестеров и быть включеной в программу учеия на реальных уязвимых примерах веб-приложений с уязвимостями, что сложно найти и помагать им в их профессиональном обучению.

ВЕБ-ПРИЛОЖЕНИЕ, JAVASCRIPT, PHP, XSS/CSRF УЯЗВИМОСТИ, БЕЗОПАСНОСТЬ ИНФОРМАЦИОННЫХ И КОМУНИКАЦИОННЫХ СИСТЕМ

ABSTRACT

The work includes 72 pages, 24 figures, 8 tables and 20 literary references.

The aim of this qualification is to secure web-applications by creating a tool that may create and eject variety of vulnerabilities in existing site or application. This tool has to inject vulnerabilitis wich are hard to detect, using vulnerability free scanners.

The object of researches is vulnerabilities, vulnerability scanners, and information data-flows, their processing in order to spot vulnerable places.

The subject of research is a security of php code in applications and web-sites.

The results are presented in the form of a table and method that demonstrates how chosen for analysis applications are secured. The results can be used in the improvement of QA testers' skills.

The results can be used as a manual for penetration testers to develop their revealing methods. This works can be used as a manual for penetration testers and can be included in teaching methods with real vulnerable web-applications and help them in their professional study.

FRAMEWORK, JAVASCRIPT, SINGLEPAGE APPLICATION, SECURITY OF INFORMATION AND COMMUNICATION SYSTEMS

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

JavaScript — популярний мова програмування сценаріїв, яка виконується у веб-браузері на стороні клієнта. Використовується для створення динамічного вмісту веб-сторінки

Cross-Site Scripting (XSS) — атака, що змушує веб-застосунки взаємодіяти з наданими користувачем даними як з виконуваними скриптами в веб-браузері користувача. При успішному виконанні атаки зловмисник може отримати доступ до всього контенту веб-браузера (Cookie, історію, версія програми тощо).

CSRF (Cross Site Request Forgery) – вид атаки, спрямований на відвідувачів веб-застосунку, що використовує недоліки HTTP протоколу.

Сканер веб-застосунків – це засоби моніторингу та контролю, за допомогою якого можна перевірити комп’ютерні сеті, окремі комп’ютери і встановлені на них програмні засоби на наявність у них проблем з захищеністю.

Cookie — невелика кількість даних, переданих веб-сервером, на сторону веб-клієнта, які можуть бути збережені і отримані пізніше. Зазвичай cookie використовуються для відстеження стану користувача під час користування веб-застосунку.

Атака — детально підібраний набір дій, які, в разі успіху, призведуть або до пошкодження ресурсів веб-застосунку або до небажаної операції.

ВСТУП

Сьогодні існує велика кількість веб-застосунків, більшість людей у розвинених країнах щодня використовують велику їх кількість. Однак багатьом розробникам бракує навичок написання безпечного коду без уразливостей. Через це, багато сучайсних сайтів і застосунків містять уразливості. Вони дозволяють зловмисникам викрадати конфіденційні дані від користувачів веб-застосунків. Щоб зменшити кількість уразливостей, можна здійснити тестування, щоб спробувати виявити, а потім виправити вразливості до того, як ці діри у бузпеці коду будуть використані. Але перш за все, щоб тестер міг виявляти можливі вразливості в веб-застосунках, він повинен знати, що шукати. Тому він повинен бути навченим різним методам, для пошуку вразливостей. Загальним методом є тестування на проникнення.

Тестування проникнення у веб-застосунках здійснюється шляхом спроби атакувати і знайти вразливі місця. Один із способів підготовки пен-тестерів - це представити тестувачу веб-сайт чи застосунок і повідомити про пошук і виявлення вразливостей у ньому.

Важливо, щоб веб-застосунки, які використовуються для навчання, були дещо реалістичними з точки зору функціональності і структури, тому вразливості в результуючих прогаммах автоматичного перетворювача вразливостей приховані в реалістичних місцях. Веб-програми також повинні бути різними з точки зору того, які уразливості містять, і як вони можуть бути використані. Тому що якщо одна й та ж веб-програма використовується двічі, тестери проникнення, які пройшли навчання на цьому веб-застосунку, вже знають, як використовувати в ньому уразливості. Таким чином, ці тестери нічого нового не навчаться. З цієї причини необхідно мати велике сховище різних веб-застосунків і їх варіацій, що містять різні типи вразливостей. Побудова та підтримка веб-застосунків та їх варіацій вручну є дорогою та трудомісткою.

Тестування на проникнення також може бути виконане автоматично за допомогою сканерів уразливостей. Використання сканерів уразливостей - це швидкий спосіб виявлення вразливостей і може використовуватися у поєднанні з ручним тестуванням на проникнення. Часто тествувальники використовують такі інструменти. Докладніше про сканери уразливості веб-застосунків читайте в розділі 2. Щоб дізнатися на які типи уразливостей потрібно зосередитися тестувальникам, треба знати, які типи сканер уразливості не може виявити. У розділі 2 буде обговорено, які типи вразливостей сканеру уразливостей важко помітити.

Актуальність роботи зумовлюється тим, що в наш час використання веб-застосунків є невідємною частиною, але важливо пам'ятати що їх не достатня захищеність та уразливості можуть призвести до втрати конфіденційних данних та коштів. У дані роботі представляється спосіб підвищення результативності роботи QA тестувальників на знаходження уразливостей у веб-застосунках.

Метою і завданням роботи є створення інструменту, що може вводити варіації вразливостей у існуючий сайт чи застосунок. Цей інструмент повинен мати можливість вбудовувати уразливості, які складно виявити сканерами веб-застосунків (що розміщені у вільному доступі).

Об'єктом дослідження є уразливості, сканери уразливостей та інформаційні процеси при потоці даних та їх обробка для знаходження не захищених місць.

Предмет дослідження — в якості предмету дослідження розглядаються захищеність php коду у застосунках та веб-сайтах.

Методи дослідження – у роботі використовувався метод статичного аналізу на чистоту та побудови абстрактного синтаксичного дерева.

Практичне застосування роботи полягає у вдосконаленні способів навчання тестувальників. Дана робота може бути теоретично посібником для пен-тестерів і бути включеною у програму навчання на реальних вразливих

прикладах веб-застосунків із вразливостями, які складно знайти і допомагали їм у їх професійному навчанні.

Наукова новизна одержаних результатів – в даній роботі проаналізовано дві найбільш розповсюджені уразливості веб застосунків на даний час та розроблено методику і інструмент по їх автоматичному впровадженні і створенні приближених до реальних ситуацій вразливого коду що удосконалить способи навчання та практики кваліфікації спеціалістів по тестуванню програмного забезпечення та його подальший захист.

Практичним значенням можуть бути використання в пізнавальних та тренувальних цілях, що несуть виключно інформаційно позитивний характер та розвиток професійних навичок написання захищеного якісного коду.

Обмеження:

- Серед усіх можливих веб-вразливостей, лише Міжсайтовий скриптинг(XSS) та Міжсайтова підробка запитів(CSRF) будуть розглянуті.
- Інструмент буде підтримувати ін'єкційні уразливості у вихідних файлах PHP. Інші мови не будуть розглянуті.

1 АТАКИ НА ВЕБ-ЗАСТОСУНКИ

Веб-застосунок - це програма, яка запускається у веб-браузері. Зазвичай джерело веб-застосунку розміщується на веб-сервері та веб-браузер робить запит на нього. Веб-сервер відповідає на запит веб-сторінкою. Веб-сторінка, яка відправляється назад, записується у підтримувані браузером мови, такі як HTML, CSS і JavaScript.

Проте сервер може створити веб-сторінку, використовуючи і інші мови на стороні сервера. Найбільш поширеною серверною мовою є PHP, який відповідно до компанії W3Techs використовується більш ніж 80% всіх веб-сайтів, які W3Techs проаналізували [19]. При використанні веб-застосунку, часто треба треба передавати деякі данні, один із способів це ввід інформації в поля input чи в рядку запиту URL-адреси (Параметри GET), за параметрами POST або в cookies [20] [11]. Ще один спосіб передачі даних у веб-застосунок здійснюється шляхом надсилання файлів [15]. Дані, що надаються користувачем, часто використовуються веб-застосунком для створення динамічних сторінок, і їх відображенні користувачеві. Наприклад, у пошуковій системі, користувач вводть пошуковий запит. Пошукова система відобразить резутати пошуку разом із веб-сторінками, які пов'язані з цим пошуковим запитом. Однак веб-застосунки часто містять вразливості, і цей розділ описує різні їх типи і те, як ці уразливості можуть бути експлуатовані

1.1 Міжсайтовий скриптинг (XSS)

Якщо веб-застосунок містить XSS-уразливості, зловмисник може виводити шкідливі данні на веб-сторінці. Зловмисно введені дані можуть містити JavaScript, який виконується, коли хтось відвідує веб-сторінку. Це означає, що зловмисник може виконувати довільний JavaScript на веб-браузері жертви, якщо йому вдасться змусити її відвідати заражену сторінку.

Шкідливий JavaScript матиме ті ж самі привілеї, що і будь-який інший JavaScript код веб-застосунку. Іншими словами, цей код матиме доступ до всіх даних, пов'язаних з уразливим доменом веб-застосунка, наприклад, файли cookie та конфіденційна інформація. Результат успішної атаки може призвести до того, що зловмисник вкраде чутливі данні, чи буде маніпулювати вмістом веб-сторінки[16]. У 2012 році компанія по розробці програмного забезпечення Symantec виявила, що XSS є найбільш поширеною вразливістю, знайденою на веб-сайтах [12], яку можна успішно експлуатувати, а CWE / SANS поставила XSS на 4 місце зі списку своїх *25 найнебезпечніших програмних вразливостей* [1].

Проблема полягає в тому, що веб-застосуок приймає вхідні данні від користувача, які пізніше використовує як у вихідних на веб-сторінці, без належного очищення введеного користувачем вхідного тексту. При фільтрації вхідних даних, відбувається перевірка на те, чи безпечно використовувати цю інформацію у подальшому виводі. Це може бути зроблено шляхом перевірки того, що що на вхід були подані прийнятні значення або маніпулюючи входом. Input полем можна маніпулювати таким чином, що навіть якщо вхід містить шкідливий JavaScript код, то він не буде виконаний. Наприклад, введені символи може бути закодоваі, так що коли кодований вхід виводиться на сторінку, браузер буде обробляти вихідні дані як текст, навіть якщо вхідні містили HTML теги, що здатні змінити структуру веб-сторінки.Прикладом такого кодування є кодування спеціальних символів у вхід користувача до об'єктів HTML [13].

Розглянемо наступний приклад:

```
<script>alert('XSS')</script>
```

При кодуванні за допомогою функції **htmlspecialchars** і флагу **ENT_QUOTES** на виході отримаємо:

```
&lt;script&gt;alert(&#039;xss&#039;)&lt;/script&gt;
```

Закодоване представлення данних цілком безпечно використовувати у веб-застосунку після їх отримання від користувача, бо браузер не буде розглядати їх як тег скрипта, лише текст що треба відобразити. Іншим способом фільтрації входу є використання **Blacklist** чи **Whitelist** фільтрів [13].

XSS представляється чотирма різними видами: **Відображені**, **Збережені**, **DOM** та **На основі мутації** [16][9]. Ці види описані в 1.1.1-1.1.4 нижче.
Важливо пам'ятати що деякі приклади у цьому розділі можуть не працювати в сучасних браузерах через методи захисту, як наприклад *Reflective XSS Protection* і кращі методи фільтрації зі сторони клієнту [3]. Одна, веб-застосунки все ще залишаються вразливими до XSS атак, залежно від браузера жертви.

Відображені атаки

Веб-застосунок, вразливий до XSS використає вхідні данні, отримані від користувача, без попередньої перевірки на достовірність та фільтрації [16]. На рисунку 1.1 зображені кроки відображеного типу атаки.

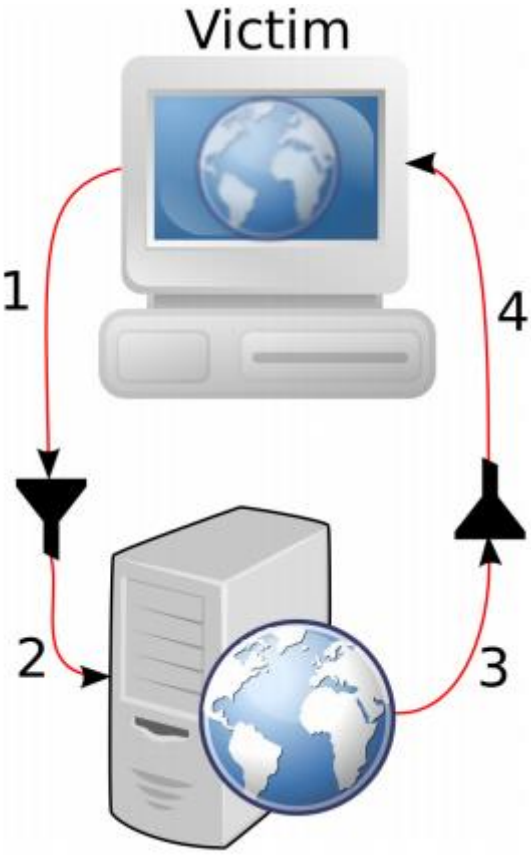

Рисунок 1.1 - Кроки відображеної XSS атаки

Жертва заповнє поля, вводячи дані, при запиті сторінки. Потім, веб-застосунок формулює вихідний запит назад жертві, який містить ті ж вхідні дані і відсилає їх.

Зловмисник може виконати відображену атаку XSS на веб-сторінці коли приймаються дані в параметрі GET для створення URL-адресу, що містить шкідливі дані. У URL-адресі зловмисник може додавати строчки кода, що містять JavaScript у параметрі GET, а потім змушує її перейти по новоствореному URL-запиті. При відвідуванні цього URL, виконується запит до вразливої веб-сторінки отримати данні, вона відповідає зі встроєним шкідливим кодом, що буде виконано на стороні клієнту.

На рисунку 1.2 показано сторінку, що є вразливою до відображеної атаки XSS. Значення параметра GET username виводиться на сторінку без фільтрації. Якби зловмисник створив URL-адресу:

```
http://target.com/?username=<script>alert("XSS")</script>
```

Тоді він одурачив би жертву перейти по URL, значення змінної *username* **(<script>alert(”XSS”)</script>)** передалося б в застосунок, а він би в свою чергу відповів би зі сторінкою, що має значення *username* і скрипт-кодом, який браузер потерпівшого обов’язково виконав. JavaScript функція *alert* виконається і відобразить модальне віконце з текстом «XSS».

Але насправді, мжна виконати не лише *alert* , а й будь-який довільний скрипт.

```
if (isset($_GET["username"])) {
    echo "Welcome ", $_GET["username"];
} else {
    echo "<form>";
    echo "<input name='username'>";
    echo "<input type='submit'>";
    echo "</form>";
}
```

Рисунок 1.2 - Приклад сторінки, вразливої до відображеного типу XSS атаки

Збережені XSS атаки

Якщо веб-застосунок вразливий до збережених XSS атак, зловмисник може зберегти зловмисні дані, котрі пізніше будуть використані як вихідні в веб-застосунку.

Дані можуть зберігатися в будь-якому місці програми, наприклад, в базі даних. Веб-програма містить сторінку, де зберігаються ці збережені дані. Збережені дані не фільтруються, перш ніж будуть збережені до того, як використатися на сторінці. Коли користувач робить запит, сторінка, що використовує збережені дані як вихідні дані в застосунку, поверне йому їх у відповідь [16]. На рисунку 1.3 показано збережену XSS-атаку. Зловмисник надає дані веб-застосунку а він зберігає їх в базі даних. Коли жертва відвідує заражену

сторінку, дані витягуються з бази даних і виводяться у відповідь користувачу та виконуються.

Щоб зрозуміти можливі наслідки такої уразливості, подумайте про онлайн-форум. На форумі користувачі можуть зареєструватися і вибрати ім'я користувача. Зареєстроване ім'я користувача зберігається в базі даних. Форум буде мати сторінку, де відображаються всі зареєстровані користувачі. Приклад такої сторінки показаний на рисунку 1.4. Якщо ім'я користувача подається, коли реєстрація не є належним чином відфільтрована, перед збереженням в базі даних. зловмисник може вибрати ім'я користувача, яке призведе до виконання JavaScriptкоду, кожен раз коли воно виводиться на сторінку. Наприклад, наступне ім'я користувача: **<script>alert(”XSS”)</script>.** Коли воно пізніше витягується з бази даних і виводиться на сторінці цей код буде виконано і відобразиться діалогове вікно сповіщення для користувача. В реальних ситуаціях встраюється довільний JavaScript замість відображення діалогового вікна сповіщення.

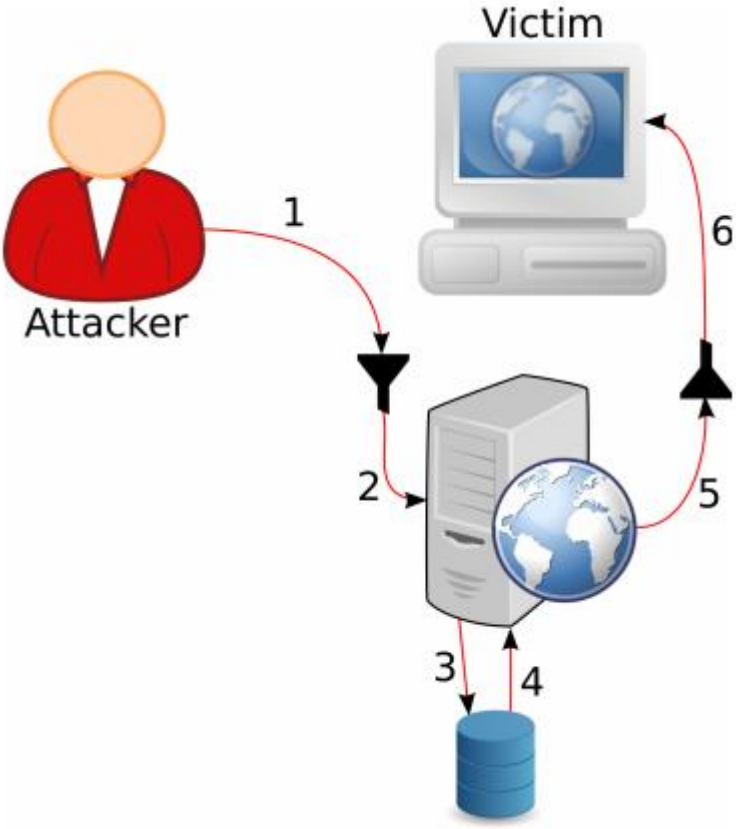

Рисунок 1.3 - Кроки в збереженій XSS атаці

```
$users = [users from database];
echo "All users: ";
foreach ($users as $user) {
    echo $user["username"], ",";
}
```

Рисунок 1.4 - Приклад сторінки, вразливої до збереженого типу XSS атаки

Об'єктна модель документа (DOM)

Атаки на основі об’єктної моделі документа є дуже схожі до збережених. Різниця в тому, що дані не відсилаються серверу, все відбувається на стороні браузера клієнта. Це значить веб-застосунок не може перевірить і профільтрувати данні на стороні серверу. Тому це треба робити з клієнту [16].

На рисунку 1.5 показаний тип XSS DOM атаки. Як це видно, данні ніколи не покидають брузера жертви.

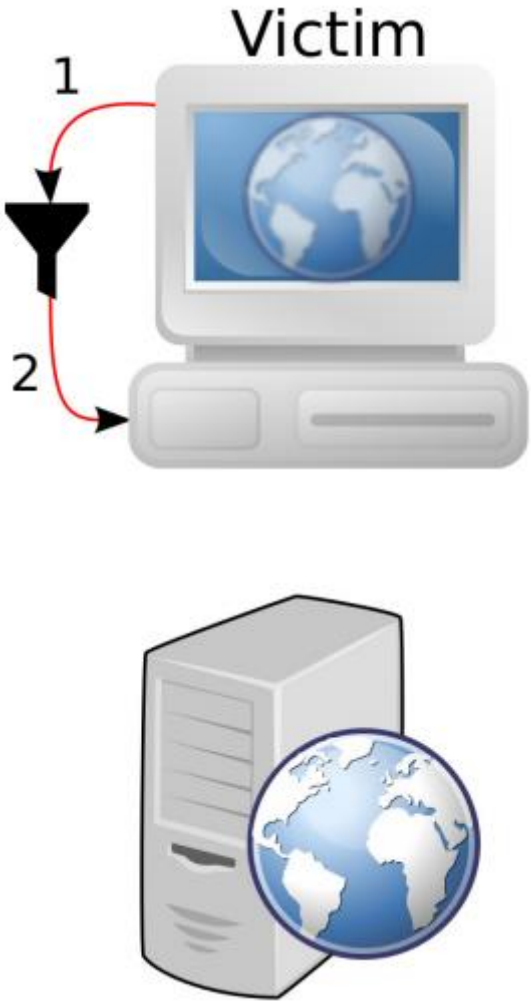

Рисунок 1.5 - Кроки DOM XSS атаки

На рисунку 1.6 GET параметр змінної **username** виводиться на сторінку використовуючи JavaScript на клієнтській стороні в браузері. Якщо сконструювати такий же URL як і в прикладі для **відображеної** атаки з 1.1.1,

```
http://target.com/?username=<script>alert("XSS")</script>
```

І потім змусити жертву перейти по ній, на сторінці виведеться **<script>alert(”XSS”)</script>.** JavaScript виконається і виведеться діалогове вікно. Але, як і в збережених типах атак, може замість цього коду, може бути будь який інший.

```
<html><head></head>
<body>
  <script type="text/javascript">
    var usernamePos = window.location.search
        .indexOf("username=");
    if (usernamePos >= 0) {
        var username = window.location.search
            .substr(usernamePos + 9);
        document.write(username);
    }
  </script>
</body>
</html>
```

Рисунок 1.6 - Приклад сторінки вразливої до DOM XSS

На основі мутації

Атаки, що базуються на мутаціях (**mXSS**) походять від з факту що веб-браузери автоматично намагаються виправити не валідний HTML код.

А робить він це мутуючи не валідний HTML у валідний HTML. Тому HTML що здається нешкідливим, може змінюватися і виконувати JavaScript. Оскільки HTML спочатку здається цілком безпечним, то може обійти багато типів фільтрації в веб-браузері і на сервері [9]. Річ що викликає мутації - використання JavaScript властивості innerHTML, що за визначенням вставляє новий вміст на сторінку. На рисунку 1.7 показано етапи атаки mXSS.

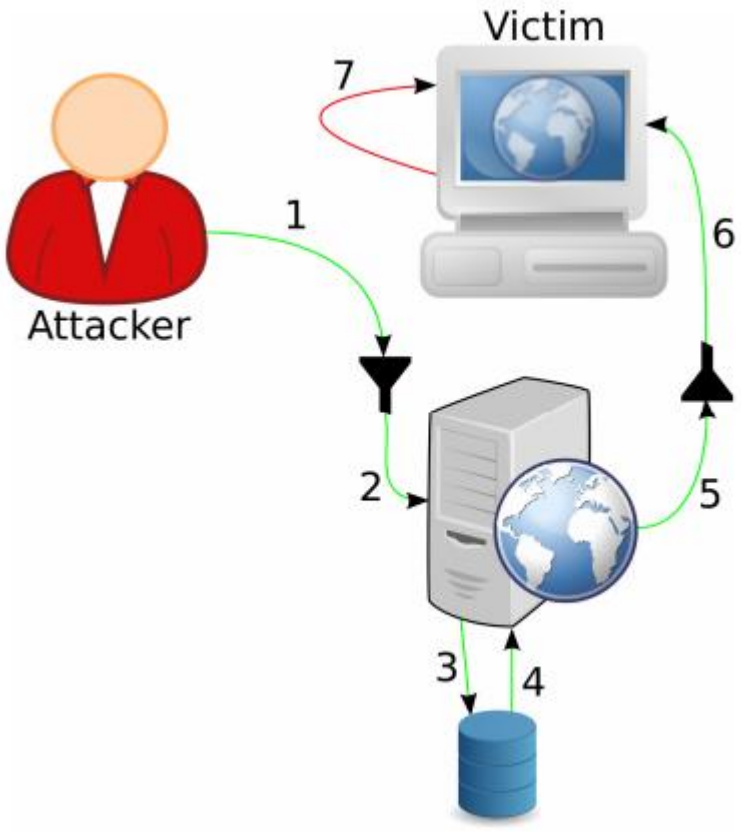

Рисунок 1.7 - Кроки атаки XSS на основі мутації

Зловмиснику вдається зберігати, здавалося б, нешкідливі дані, що містять зловмисні скрипти у веб-додатку. Коли потерпілий відвідує веб-додаток, то отримує, здавалося б, нешкідливі дані. Однак, коли клієнт використовує їх, веб-браузер жертви змінює дані і виконує JavaScript, який зловмисник включив до збережених даних.

Приклад того, як веб-браузер(в даному випадку Internet Explorer 8) мутує невірний HTML, коли введені дані:

```
<s class="">hello&#x20;<b>world</b>
```

Через не валідний HTML, браузер змутує його у валідний код і результатом буде:

```
<S>hello <B>world</B> </S>
```

Браузер видалив пустий атрибут class, усі теги перетворилися у верхній регістр е, ** ** замінився на пробіл і з'явився закриваючий тег <S>. Дана мутація не

змутувала ні в який зловмисний код, але на рисунку 1.8 сторінка вразлива до mXSS атаки

```
<script>
function post () {
    messageElement = document.getElementById("message");
    imageElement = document.getElementById("image");
    var alt = document.getElementById("inAlt").value;
    var message = document.getElementById("inMessage").value;
    var img = '<img src="test.jpg" alt="' + alt + '">';

    imageElement.innerHTML = img;
    messageElement.innerHTML = message;
    messageElement.innerHTML += imageElement.innerHTML;
}
</script>

<div id="image"></div>
<div id="message"></div>

message: <input id="inMessage"></input><br>
alt: <input id="inAlt"></input><br>
<button onclick="post()">Post</button>
```

Рисунок 1.8 - Приклад сторінки, вразливої до атаки XSS на основі мутації

Якщо передати “onload=alert(’XSS’) в alt атрибут, то при натисканні кнопки <img> тег матиме вигляд:

```
<img src="test.jpg" alt="‘‘onload=alert(’XSS’)">
```

Дані збереглися в атрибуті alt і не будуть виконані. Однак, коли браузер використовує JavaScript властивість **innerHTML** щоб вставити міст в message div тег, то HTML змутує у

```
<IMG alt="" src="http://example.com/xss/test.jpg"
    onload=alert('XSS')>
```

Таким чином браузер виконає JavaScript код і відобразе діалогове вікно.

1.2 Міжсайтова підробка запиту (CSRF)

CSRF атака складається з трьох частин: жертва, довірений сайт де жертва авторизована і зловмисник [17]. На риснку 1.9 зображено приклад CSRF атаки.

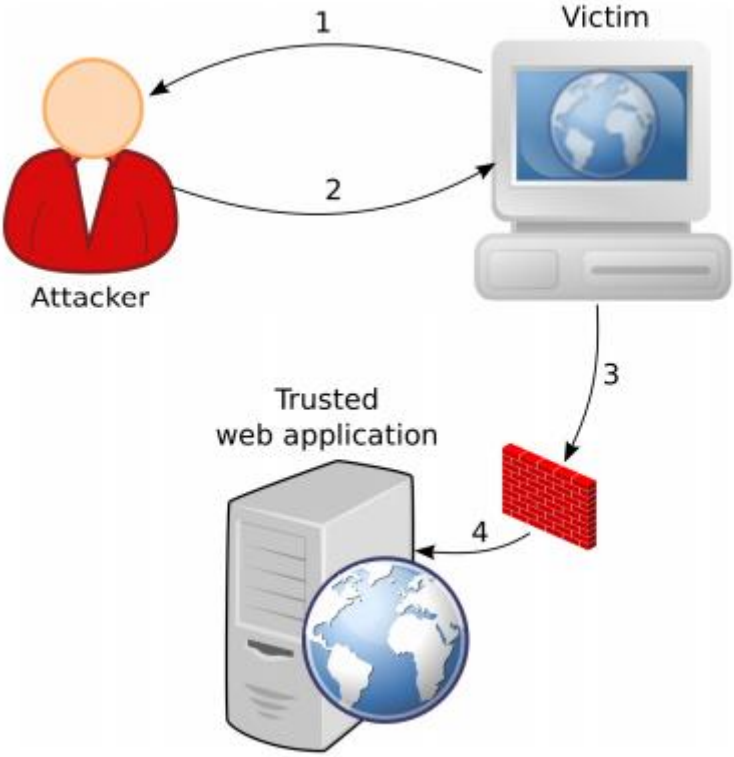

Рисунок 1.9 - Кроки в CSRF атаці

Жертва авторизована на довіреному сайті відвідує веб-сторінку, контрольовану зловмисником. При переході на неї, браузер жертви отримує повідомлення про відправлення запиту на довірений веб-застосунок. Веб-сторінка може сказати переглядачу надіслати запит на інший веб-застосунок, наприклад, вставивши наступний тег img:

```
<img src="http://trusted-site.com/change_password.php
    ?newpassword=pwned">
```

Веб-браузер жертви намагатиметься отримати зображення, тобто надіслати запит до URL-адреси зображення. Тому що зображення знаходиться на довіреному домену веб-програми, веб-браузер жертви буде включати всю

інформацію, яку він пов’язав із цим доменом. Це включає файли cookie, які веб-додаток використовує для перевірки автентичності жертви.
Тому, коли довірентй веб-додаток отримує запит від жертви, сеанс буде аутентифіковано. Для довіреного веб-сайту, запит виглядатиме як звичайний нормальний запит.

```
if (isset($_SESSION["USERID"])) {
    $currentUser = $_SESSION["USERID"];
    $newPassword = $_GET["newPassword"];
    update_user_password($currentUser, $newPassword);
    echo "Password updated";
}
```

Рисунок 1.10 - Сторінка вразлива до CSRF

У цьому прикладі сторінка, що запитується на довіреному веб-застосунку змінює пароль жертви (рисунок 1.10 має копію вихідного коду сторінки).

Після того, як потерпілий зробив запит довіреному веб-застосунку, пароль жертви буде змінено на pwned.
Проблема уразливіості CSRF полягає в тому, що в довіреносу веб-сайту бракує способу перевірити, чи жертва навмисно відправила запит. Якщо така перевірка відсутня, зловмисник може обдурити жертву, щоб зробити ненавмисний запит до довіреного веб-застосунку. Може зробити довірену дію від імені жертви оскільки зловмисник може змусити потерпілого відправити запит до довіреного веб-застосунку і важко перевірити які насправді наміри цього запиту [7].

CSRF є 12ю у списку «*Топ 25 найбільш небезпечних уразливостей*» за версією компанії розробки та забезпечення безпеки програмнго забезпечення CWE/SANS [1]. Основний існуючий метод захисту від цього типу атаки – використання токену CSRFtoken [6][5][17]. Цей метод розподілений на 3 частини:

Generate - Токен, котрий дуже складно вгадати генерується у веб-застосунку на сервері. Потім, він зберігається десь де веб-застосунок може пізніше його отримати, наприклад в cookies клієнта

Include - Токен відправляється на браузер клієнта, щоб бути доступним при виконанні дії авторизації. Наприклад як поле в формі відправлення, що авторизувати користувача.

Guard - Коли запит на виконання авторизованої дії отримано, веб-застосунок перевірить переданий клієнтом токен, тобто він має співпасти з серверним, що був не давно згенерований. Якщо вони не співпадають то дія не буде виконана.

Висновки до розділу 1

У даному розділі було розглянуто XSS та CSRF атаки. Було проаналізовано їх різновидність та детально описано функціонал та можливі випадки застосування.

Сформульовано основні принципи, та з’ясовано їх функціонування на стороні серверу та клієнта, розглянуто аналіз компаній-розробників та за хисту програмного забезпечення, як W3Techs стосовно небезпеки XSS атаки в даний час.

2 АНАЛІЗ СКАНЕРІВ ВЕБ-ЗАСТОСУНКІВ

Сканер веб-застосунків – це засоби моніторингу та контролю, за допомогою якого можна перевірити комп'ютерні сеті, окремі комп'ютери і встановлені на них програмні засоби на наявність у них проблем з захищеністю. Використовуються для автоматичного сканування веб-застосунків та пошуку у них уразливостей. Такі сканери можуть лише посилати запити на застосунок і перевіряти відповідь, але у них не має доступу до перевірки коду [8]. Іншим словами, сканери веб-застосунків можуть взаємодіяти з застосунком чи сайтом лише в той спосіб що робить зловмисник.

Взагалі кажучи, сканери уразливості містять три модулі/етапи: *crawler(пошук)*, *attacker(атака)* та *analysis(аналіз)* [8]. При початку сканування сканеру надається ряд URL-адрес. Модуль *crawler* відвідує їх та збирає всі можливі URL на сторінках веб-застосунку. Потім він відвідає усі новозібрані адреси, щоб знайти ще більше URL-адрес. Це буде продовжуватся поки модуль сканера не зібрав якомога більше доступних йому веб-сторінок. Усі вхідні точки у веб-застосунок (наприклад, поля input, GET праметри) також збираються під час етапу *пошуку*. Коли модуль сканера завершиться, він передасть усі дані, які зібрав (доступні веб-сторінки та вхідні точки) до модуля атаки(*attacker*). Модуль атаки зконструює вектори атаки на основі отриманих даних від модуля *пошуку*. Вектори атаки містять вхідні значення, які можуть викривати уразливості. На цих векторів атаки, атакуючий модуль буде надсилати запити до веб-програми. Всі відповіді, що отримуються з веб-застосунку надаються до модуля *аналізу*. Для кожної відповіді від застосунку модуль аналізу намагається визначити, чи містяться будь-які сліди, що вказують на наявність уразливості. Якщо веб-сканер знаходить вектор атаки, який генерує відповідь, що вказувала на наявність уразливості, то позначить атаковані веб-сторінки як уразливі і повідомить про те, який саме вектор атаки викликав уразливість.

Існує взагалі велика різновидність сканерів веб-застосунків із різними можливостями. OWASP та The Web Application Security Consortium мають список потужних сканерів [14][18]. З обох проектів було зібрано список сконерів веб-уразливостей, що можна скачати у вільному доступі. У таблиці 2.1 цей список відображено, разом з інформацією чи підтримується пошук XSS та CSRF уразливості. Усі вони є безкоштовними та вільнодоступними.

Таблиця 2.1 - Список безкоштовних сканерів веб-застосунків

Назва	Платформа	XSS	CSRF
andiparos	OS X, Linux, Windows	Так	Ні
Grabber	Python	Так	Ні
OWASP Zed Attack Proxy	Windows, Linux, OS X	Так	Так
Paros	Windows	Так	Ні
Powerfuzzer	Python	Так	Ні
Skipfish	Linux, OS X, Windows	Так	Ні
w3af	Python	Так	Так
Wapiti	Python	Так	Ні

2.1 Труднощі пошуку вразливостей

При аналізі сканерів, було виявлено, що результати сканування не правдоподібні у багатьох випадках, коли зловмисні дані спершу зберігаються у веб-застосунку і потім використовуються у виводах на інших сторінках, викликаючи ці вразливі дані [4]. Відповідь на запит сканера не міститиме жодного натяку наявності уразливості, бо результат атаки спрацює на зовсім іншій сторінці.

Аналогічні результати виявив професор каліфорнійського інституту Doup´e [8]. Приклад атаки що спершу зберігає данні у збереженому типі XSS атаки, обговорений у 1.1.2.

Сканерам уразливостей не просто знайти CSRF без похибок(коли знаходиться багато результатів, що насправді не є вразливими, а сканер їх все рівно знайшов). Це через те, що складно оцінити який запит має бути захищеним від CSRF [2]. Один зі способів спробувати знайти цей тип атак полягає у записі кожного запиту і коли сканування завершено, то пробувати посилати окремо запити по відібраним записам ймовірних вразливостей. Якщо повторні результати були успішними, то можна сказати з більшою ймовірністю про наявність CSRF уразливості. Але знову ж таки, результат не буде 100 процентовим, бо у багатьох застосунках не має необхідності на деяких сторінках вводити захист проти CSRF.

Висновки до розділу 2

У цьому розділі було розглянуто сканери веб-застосунків, описано їх етапи роботи та перевірки уразливих місць застосунків, також описано основні недоліки та висвітлено список основних сканерів на ринку, їх властивості. Обговорили результати тесту перевірки на збережений тип XSS атак, на прикладі статті каліфорнійського професора.

3 РОЗРОБКА ТА ВПРОВАДЖЕННЯ АВТОМАТИЧНОГО ПЕРЕТВОРЮВАЧА КОДУ

Як зазначено у вступі, метою цього дипломного проекту було створення програмки для автоматичного перетворення коду та встроєння уразливостей у веб-застосунки. Ціль створеного полягає у більш якісній підготовці тестувальників, особливо пен-тестерів та оцінки сканерів виявлення уразливостей. Інструмент має бути розроблений і реалізований і більш конкретно повинен мати можливість:

- Встроювати XSS та CSRF вразливості в PHP код
- Зробити процес, настільки це можливо, автоматизованим
- Генерувати діри, що інфікують більше одного системного файлу
- Вводити варіації одного і того ж типу уразливості, щоб не було однакових варіантів
- Реалізовувати вразливості, що складно виявити простими вільнодоступним сканером
- Діри, створені генератором мають підлягатися хакерському взлому

Для цього було створено **VulnerableCodeGen**. У розділі 3.1 описується, як передбачається використовувати цей застосунок. Розділ 3.2 описує категорії та типи уразливостей, які ця программа може і не може обробити. Розділи 3.3 і 3.4 описують внутрішню роботу **VulnerableCodeGen**, як він генерує та встроює XSS і CSRF уразливості в уже існуючий код веб-застосунка чи сайту.

3.1 VulnerableCodeGen

У цій секції буде описано етапи створення та впровадження програми **VulnerableCodeGen,** що задовільняє умовам, описаним вище. Весь проект складається з 2х етапів: підготовка проекту та встроєння уразливостей в код.

Рисунок 3.1 - Послідовність етапу підготовки

Розділи 3.1.1 і 3.1.2 описують, як працює VulnerableCodeGen і як використовувати VulnerableCodeGen на цих етапах.

3.1.1 Підготовка проекту

Перш ніж VulnerableCodeGen зможе вбудувати вразливості в проекті, спочатку слід підготувати вихідний код. Це пов'язано з тим, що важко в цілому проекті зробити стастичний аналіз для визначення того, як працює воркфлов веб-застосунку, тобто як программа керує данними, приймає та віддає їх, які запити робить і тд. Особливо важко статистично знайти потік даних, якщо input і output знаходяться в різних файлах, і дані можуть зберігатися в базі даних між input і output программи. Етап підготовки обмежує кількість вихідного коду, який VulnerableCodeGen повинен проаналізувати, щоб ввести вразливість. Запланований робочий процес під час підготовки проекту можна побачити на рисунку 3.1. Веб-застосунок має бути спочатку проаналізовано та підготовлено вручну. Коли знайдено місце розташування, де може бути введена вразливість, треба розмістити ключові слова в коді, щоб сказати VulnerableCodeGen про

аналіз цієї частини вихідного коду. Приклад вихідного коду, підготовленого для аналізу VulnerableCodeGen, можна побачити на рисунку 3.2.

Коли між ключовими словами/ * VulnerableCodeGen * / на одній строчці коду знайдено об'єкт конфігурації JSON, тоді все до останнього ключового слова / * / VulnerableCodeGen * /, буде блоком коду, який VulnerableCodeGen проаналізує. Об'єкт конфігурації включає в себе ідентифікатор, який з'єднує блок коду з певною вразливістю. Коли введена вразливість, визначена ідентифікатором, кожен блок коду з однаковим id аналізується, . Об'єкт конфігурації також містить розміри (деталі в розділі 3.2), в яких ця частина уразливості може змінюватися. Залежно від типу уразливості об'єкт може містити більше атрибутів. Після створення вихідного коду необхідно створити файл конфігурації для всього проекту. Приклад такого файлу можна побачити на рисунку 3.3. Файл містить об'єкт JSON, що містить ім'я проекту, відповідний каталог, в якому розміщені вихідні файли веб-програми, і список всіх можливих вразливостей, які можуть бути введені, та ідентифіковані ідентифікатором. Кожна вразливість має тип, опис, список файлів і опцій. Об'єкт options містить різні ключі залежно від типу уразливості. У розділах 3.3 і 3.4 буде обговорено об'єкт опцій для XSS і CSRF уразливостей. Обмежуючи код, VulnerableCodeGen аналізує лише ті частини, що є важливими для генерації вразливості. Таким чином, розробник може бути впевненим, що необхідні вразливості, будуть вбудовані у вихідний код.

```
{
    "name": "Phulner testproject",
    "basedir": "files/",
    "vulnerabilities": {
        "xss_echoId": {
            "description": "Echoes the user supplied id",
            "type": "xss",
            "options": {
                "input": ["GET"],
                "sanitation": ["NONE", "BLACKLIST"],
                "output": ["NORMAL_TAG"],
                "mutated": [false]
            },
            "files": [
                "welcome.php"
            ],
            "sanitationFunctions": []
        },

        "csrf_changePassword": {
            "description": "Protects the change password
                action",
            "type": "csrf",
            "options": {
                "types": ["NONE", "ONLY_POST", "COMPUTABLE"]
            },
            "files": [
                "changePassword.php"
            ]
        }
    }
}
```

Рисунок 3.2 - Приклад конфігураційного файлу для VulnerableCodeGen

3.1.2 Генерація вразливого веб-застосунку

Схема послідовності генерації вразливого веб-застосунку з підготовленого проекту показана на рисунку 3.3. Перш за все, має бути створений конфігураційний файл(приклад такого файлу показано на рисунку 3.4).

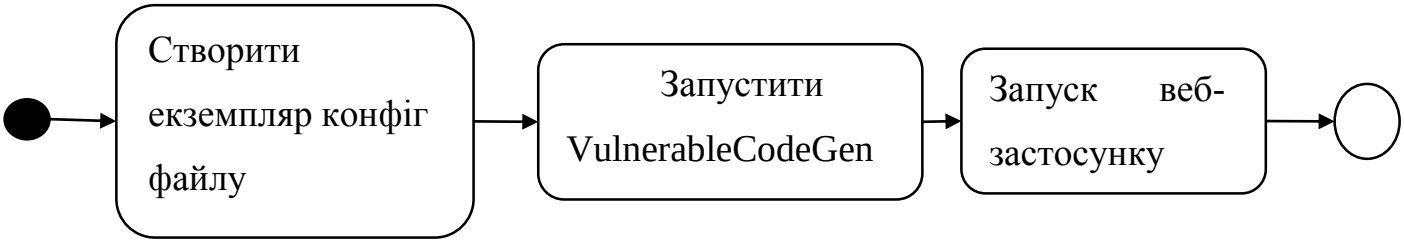

Рисунок 3.3 - Схема етапу «впровадження вразливості»

Конфігураційний файл містить б'єкт JSON, який визначає, з якого проекту VulnerableCodeGen повинен генерувати вразливий веб-застосунок, які вразливості повинні бути введені, і в яких категоріях будуть вбудовані вразливості. При запуску VulnerableCodeGen, файл конфігурації надається в якості вхідних даних. VulnerableCodeGen буде завантажувати вказаний проект і вводити вказані вразливості відповідно до файлу конфігурації. Новостворена уразлива веб-програма буде збережена по вказаному шляху в конфігурації екземпляра. Змінюючи параметри вразливостей у файлі конфігурації можна отримати новий екземпляр программи з іншими вразливостями.

```
{
    "project": "/path/to/phulner/project/",
    "out": "/destination/of/vulnerable/web/application/",
    "vulnerabilities": {
        "xss_echoId": {
            "inject": true,
            "sanitation": "NONE"
        },
        "csrf_changePassword": {
            "inject": true,
            "type": "NONE"
        }
}
```

Рисунок 3.4 - Приклад файлу конфігурацій

3.2 Категоризація

Вибрані вразливості були розділені на різні категорії, щоб розрізняти різні варіації однієї і тої ж вразливості.

Категорії також використовуються при виявленні того, по якій категорії шукають вразливості сканери при скануванні коду. Категорії були створені на основі характеристик кожної вразливості відповідно.

3.2.1 Впровадження XSS

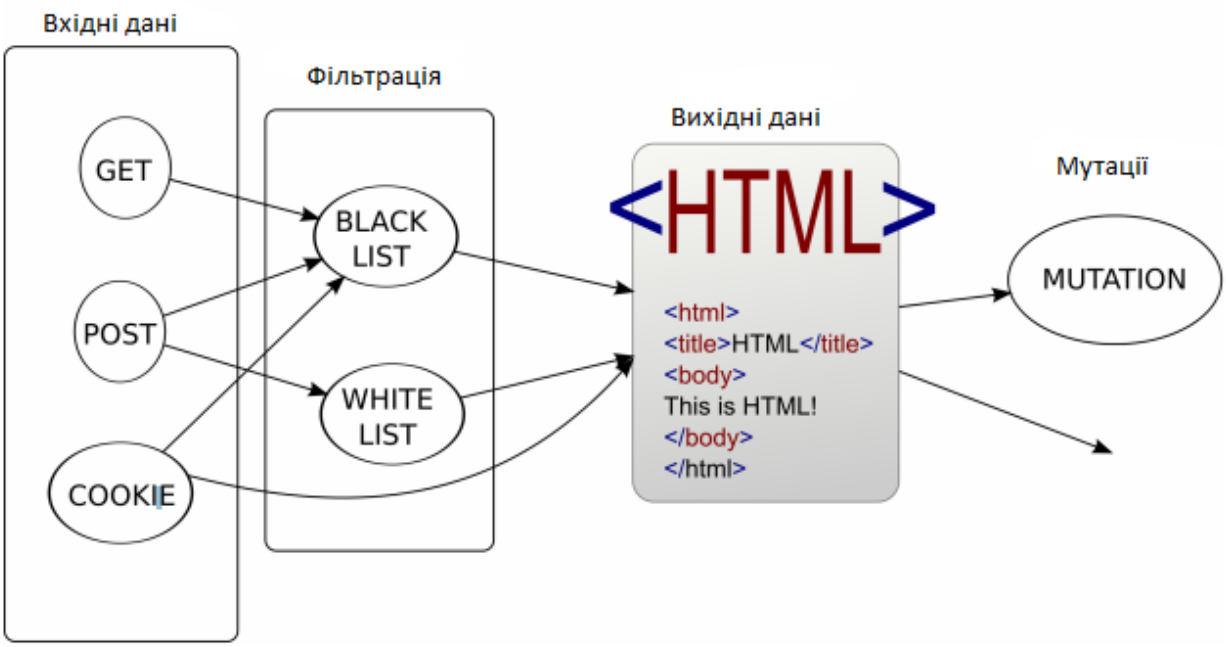

Рисунок 3.5 - Діаграмма поділу XSS вразливості на етапи її виявлення та впровадження

Перш за все, коли зловмисник намагається знайти XSS, він спочатку намагається виявити як данні вводяться, та реакцію програми на введення, тобто результат. Наступний крок полягає у дослідженні який з input, чи інших полів можна використати при атаці. Це робиться шляхом подання різних данних у

поля вводу і аналізу відповіді від серверу. Виходячи з того, як зловмисник працює при спробі виявити XSS, ми бачимо, що уразливість має певні характеристики і групуємо відповідно до них. Починається з того, що вводиться в поля. Ще вхідні дані можуть бути частково фільтровані. Після цього щось виводиться в браузер і браузер може мутувати вихід, тобто створювати різні варіації атак. На рисунку 3.5 показані описані вище характеристики. Згідно з ними я розділив впровадження вразливості XSS на чотири етапи.

1) Вхідні данні (input)

Перше, що зловмисник повинен з'ясувати, як подавати дані до програми. Введення може бути здійснене кількома способами. Ті способи, які VulnerableCodeGen може обробити, класифікуються наступним чином:

- GET

Поле вводу має параметр GET.

- POST

Параметр вводу POST

- Cookies

Параметр вводу Cookies

- Header

Поле вводу являється полем Header

- Stored

В полі відображаються данні, що були раніше, або попередньо збережені сайтом чи веб-застосунком. Наприклад, з бази данних.

2) Фільтрація

Коли способи, якими вводиться данні в поля вводу визначено, треба з’ясувати чи введені данні попередньо обробляються, перед використанням, тобто фільтруються. Цей крок виконується для очищення вводу від частин

тексту, чи данних що ніяк не мають входити. Іншими словами це валідація введених полів. Відповідні різновиди цього етапу:

- **NONE**

  Вміст поля не перевіряється і не валідується зовсім

- **Black list**

  Набір символів і послідовностей внесено до чорного списку та видалено з вмісту.

- **White list**

  Є протилежним Black list. Дозволені лише певні символи та послідовності, а все інше видаляється зі вмісту

- **Кодування**

  Популярний метод зменшення ризику атак при відображенні вводу користувача полягає в кодуванні або перетворенні. Якщо застосовується недостатнє кодування або недостатньо перетворюється, шкідливі дані можуть бути все ще введені в поле.

3) Вихідні данні (output)

Залежно від того, де вміст виводиться на сторінку, довжина зловмисного коду та його вміст змінюється. Наприклад, якщо зловмисник зможе ввести вміст у тег Script, вміст буде виконано як JavaScript сцинарій. У звичайному тегі, дані не будуть виконуватися як JavaScript. Зловмисник повинен буде ввести щось, що змушує клієнта виконувати дані як JavaScript. Другий фрагмент коду в наступних прикладах показує, що зловмисник може надсилати, щоб мати змогу запускати JavaScript у кожній категорії. У наступному списку відображаються місця виводу, які VulnerableCodeGen може обробити.

Script tag

<script>HERE</script>

Данні будуть виконані як JavaScript код.

alert("XSS")

HTML Comment

<!-- HERE -->

Дописуючи закриваючу скобку для коментаря, хакер може вставити необхідний тег і виконати JS код

--><script>alert("XSS")</script><!—

Attribute name

<img HERE=x>

Вставляючи пробыл, можна відокремити атрибут і вставити будь який інший, або набір атрибутів.

onload="alert(’XSS’)" src

Tag name

<HERE >

Зловмисник сам вирішує, який тег буде використовуватися. Якщо він ще й може вирішити зміст тегу, то вкаже тег скрипта, і все, що міститься в тезі, все буде виконано. Якщо ні, то він може додати будь-який атрибут до тегу.

img onload="alert(’XSS’)" src="x

Style tag

<style>HERE</style>

Деякы браузери виконають JavaScript.

body { background:url("javascript:alert(’XSS’)") }

Normal tag

* HERE *

Зловмисник може встроъти тег скрипта. Контент всередині виконається

<script>alert("XSS")</script>

URL attribute - unquoted

<a href=HERE>

Якщо атакуючий вставить JavaScript URL і користувач клікне по ссилкі, тоді код буде виконано.

javascript:alert(’XSS’)"

URL attribute - single quotes

<a href=’HERE’>

Якщо атакуючий вставить JavaScript URL і користувач клікне по ссилкі, тоді код буде виконано.

javascript:alert("XSS")

URL attribute - double quotes

<a href="HERE">

Якщо атакуючий вставить JavaScript URL і користувач клікне по ссилкі, тоді код буде виконано.

javascript:alert(’XSS’)

Non JavaScript attribute - unquot

<input value=HERE>

Пробіл дасть можливість вставити будь-який атрибут.

x onclick="alert(’XSS’)"

Non JavaScript attribute - single quotes

<input value=’HERE’>

Одиничні лапки дадуть можливість вставити будь-який атрибут.

x’ onclick=’alert("XSS")

Non JavaScript attribute - double quotes

<input value="HERE">

Подвійні лапки дадуть можливість вставити будь-який атрибут.

x" onclick="alert(’XSS’)

JavaScript attribute - unquoted

<button onclick=HERE>

Данні будуть виконані як JavaScript код.

alert("XSS")

JavaScript attribute - single quoted

<button onclick=’HERE’>

Данні будуть виконані як JavaScript код.

alert("XSS")

JavaScript attribute - double quoted

<button onclick="HERE">

Данні будуть виконані як JavaScript код.

alert(’XSS’)

JavaScript manipulation

Контент вставлений в DOM-дерво з JavaScript кодом.

Інша локація

Інше місце для вставлення зловмисного коду, не розлянутого в цьому списку.

4) Мутації

Якщо контент вставлений у сторінку за допомогою, наприклад, властивості innerHTML, то він має бути видозміненим, як було розглянуто це у 1.1.4

- Так

Дані використовуються у такому вигляді, що певні клієнти змутують вміст контенту.

- Ні

Дані не будуть змутовані клієнтом

3.2.2 CSRF

Категорії вразливостей CSRF базуються на загальних принципах зменшення, які можуть бути недостатніми для зупинки атаки CSRF [6].

- **None(немає)**

Сервер не виконує жодних перевірок, щоб переконатися, що запит був навмисно відправлений, перш ніж виконувати аутентифікацію.

- **Only POST request (Тільки запит POST)**

Сервер приймає тільки POST-запити, щоб виконати аутентифікаційні дії. Це зупинить атаки, які вимагають від жертви натиснути на посилання (тому що генеруватиме запит GET до довіреного сервера).

- **Кілька кроків**

Якщо для аутентифікованої дії потрібно виконати більше одного запиту, зловмисник повинен виконати ряд запитів у певному порядку, щоб успішно виконати атаку.

- **Referer заголовок**

  Якщо єдина перевірка, яку виконує сервер, перед перевіркою автентичності це перевірка Referer заголовку. У цьому заголовку браузер може вказати, з якого URI виник запит [10]. Сервер може блокувати всі запити, які не походять з надійного URI.

- **Token**

  Як обговорювалося в 1.2, загальним способом захисту від атак CSRF є використання генерованого токену. Якщо наданий маркер(токен) не є випадковим, зловмисник може його обчислити.

3.3 XSS ін’єкції

```
$foo = intval('bar');
```

Рисунок 3.6 - Код, що створює абстрактне дерево на рисунку 3.7

Інструмент, розглянений в розділі 3.3 містить функціонал для аналізу і пошуку конкретних шаблонів у вихідному коді. VulnerableCodeGen використовує абстрактне синтаксичне дерево для представлення вихідного коду. Використовуючи це дерево, VulnerableCodeGen може зосередитися на розумінні логіки коду.

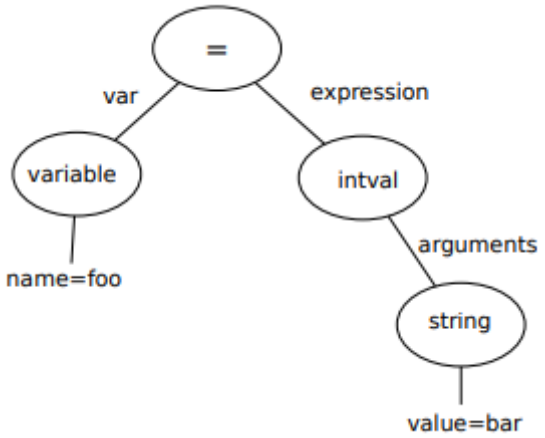

Рисунок 3.7 - Абстрактне синтаксичне дерево, що створюється під час виконання коду на рисунках 3.6, 3.8 та 3.9

```
$foo
                                =
intval("bar"    );
```

Рисунок 3.8 - Код що створює Абстрактне синтаксичне дерево на рисунку 3.7

Абстрактне синтаксичне дерево являє собою оператор, інструкції , цикли та вирази. Дерево захоплює важливу структуру вихідного коду без розуміння синтаксичних деталей, такі як пунктуація. Наприклад, коди на рисунках 3.6, 3.8 та 3.9 будуть мати таке ж абстрактне синтаксичне дерево як на рисунку 3.7. Граматика коду варіюється між прикладами але логіка в коді однакова. Абстрактне дерево також полегшує аналіз аргументів функції-виклику. Наприклад, уразливість XSS у категорії Non JavaScript attribute - single quotes. Таким чином, змінна що передана до функції htmlspecialchars може бути використана, як показано нижче:

```
$src = htmlspecialchars($userinput);
echo "<img src='", $src , "'>";
```

Бо, за замовчуванням htmlspecialchars функція не декодує одинарні кавички(‘), таким чином не місце може бути перероблене на вразливе. Якщо

зловмисник передасть строчку кода http://example.com/image.jpg’ onload=’alert("XSS"), тоді вихідний тег img буде мати вигляд:

```
<img src='http://example.com/img.png'
    onload='alert(&quot;XSS&quot;)'>
```

Що значит, що буде показане повідомлення користувачу про XSS, по закінченню завантаження картинки.

```
${'foo'} /* a block comment here with the '=' symbol */
= // line comment
intval # another type of comment
(/* another block comment including an assignment
    £foo = intval("bar") */ "bar");
```

Рисунок 3.9 - Код що створює Абстрактне синтаксичне дерево на рисунку 3.7

Для того, щоб зробити кодування одинарних лапок, константа **ENT QUOTES** має бути передана як другий аргумент [16]. Якщо змінну передали в **htmlspecialchars** з цією константою, як показано в наступному коді:

```
$src = htmlspecialchars($userinput, ENT_QUOTES);
echo "<img src='", $src , "'>";
```

І якщо зловмисник вставить попередню строчку (http://example.com/img.png’ onload=’alert(’’XSS’’)), тоді атака не матиме успіху. Це виникає через те, що **htmlspecialchars** кодує одинарні лапки, що значить що отриманий з серверу тег **img** буде мати вигляд:

```
<img src='http://example.com/img.png&#039;
    onload=&#039;alert(&quot;xss&quot;)'>
```

Оскільки одиничні лапки кодуються, атакуючий код не може вийти за атрибут src і вставити новий атрибут, який буде виконувати JavaScript. При використанні абстрактного синтаксичного дерева всі відомості про функцію виклику, такі як аргументи, будуть у стекі виклику функції. Якщо програмка обробив(розпарсив) вихідний код, потрібно більше детально розписувати логіку

щоб переконатися, що **ENT_QUOTES** передається як другий аргумент функції **htmlspecialchars**. Ще однією перевагою при використанні абстрактного синтаксичного дерева є те, що при заміні вузлів(nodes), неможливо порушити синтаксис коду веб-застосунку. Валідне абстрактне синтаксичне дерево завжди перетворюється в синтаксично правильний код, без помилок. Щоб відслідковувати, як користувацький ввод в поле input маніпулює далі з цими даними, VulnerableCodeGen використовує метод, що називається **статичним аналізом на чистоту(з англ. Static Taint Analysis).**

Змінні, що залежать від вхідних даних користувача і не фільтруються(не використовуються sanitize функції), позначені як taint змінні(брудні змінні). За допомогою статичного аналізу, VulnerableCodeGen може проаналізувати, коли користувальницький вхід фільтрується у веб-застосунку. Більш докладно про те, як цей метод працює можна прочитати у розділі 3.3.1.

Для того, щоб допомогти VulnerableCodeGen зрозуміти, як функції фільтрують змінні, існує папка зі спеціальними sanitize функціями у VulnerableCodeGen . Ці функції описують як певена функція працює у аспекті фільтрації своїх аргументів. Іншими словами, вона описує, чи taint від аргументів функції передається до значення, що повертається з функції. Де «Taint» («забруднення») - це спеціальна мітка на змінній, що в ній лежать недовірених дані, змішування довірених і недовірених даних призводить до «забруднення» результату. Недовірених даними є весь ввод - те, що прийшло з вхідного потоку, прочитано з файлів і так далі, довіреними - то, що зробила сама програма. Коли використовується функція, то вона уже знає про вразливість, яку VulnerableCodeGen в даний час намагається ввести. Залежно від різних опцій уразливостей, таких як output категорія, sanitation функція може визначити, чи буде цей taint поширюватися. Функція фільтрації для вбудованої PHP функції htmlspecialchars буде визначати що значення яке повертається з неї буде помічене міткою tain якщо: перщий аргумент помічений нею, константа ENT_QUOTES не використовується і output категорія - Non JavaScript attribute -

single quotes. Однак, якщо вихідна(output) категорія була **Normal Tag** тоді **htmlspecialchars** не буде поширювати taint з першого аргументу. Для того, щоб можна було виконати JavaScript код в звичайному тегі, треба дадати новий тег, як наприклад те скрипта, а це не можливо, бо **htmlspecialchars** кодує < and > за дефолтом. Це кодування унеможливлює створення нового тегу. Sanitaiton функція також визначає як функція може бути замінена для збереження taint аргументу в залежності від опцій уразливості. Те як це працює на практиці, буде розглянуто нижче:

```
        /* VulnerableCodeGen
        {
            "identifier":       "xss_echoId",
    "sanitation": ["NONE", "BLACKLIST"];
            "initialScope": [
                {
                    "name": "userinput",
                    "type": "variable",
                    "taint": ["USER"]
                }
            ]
        }
        */
        $id=intval($userinput
); echo $id;
        /*/ VulnerableCodeGen */
```

Рисунок 3.10 - Код, підготовлений для введення XSS вразливості

При введенні XSS уразливості VulnerableCodeGen може змінювати вразливість, базуючись на тому, що передано в sanitation властивість об'єкта(рисунок 3.10) . Підготовка при введенню XSS складається з: визначення вхідної, вихідної та мутаційної категорії; вказуючи, в яких sanitation категоріях ця уразливість може змінюватись і вказувати, які змінні позначаються як taint

на початку блок. На рисунку 3.10, показаний підготовлений код для вставки вразливості . Блок коду належить до уразливості xssechoId, і можуть бути змінені в категоріях NONE або BLACKLIST. Змінна $userinput позначена як tainted(забруднена) на початку блоку коду. Більше інформації про вразливість знаходиться в конфігураційному файлі проекту (показаний на рисунку 3.4). Конфігураційний файл проекту вказує, що вразливість, ідентифікована як xss_echoId типу XSS. Вхідний параметр - це параметр GET, а output локація - у звичайному тегі. Дані не будуть мутувати. Коли **VulnerableCodeGen** проходить по блоку кода що належить XSS, як наприклад код рисунку 3.10, він парситься в абстрактне синтаксичне дерево, використовуючу бібліотеку PHP Parser[29]. Результуюче дерево можна побачити на рисунку 3.11.

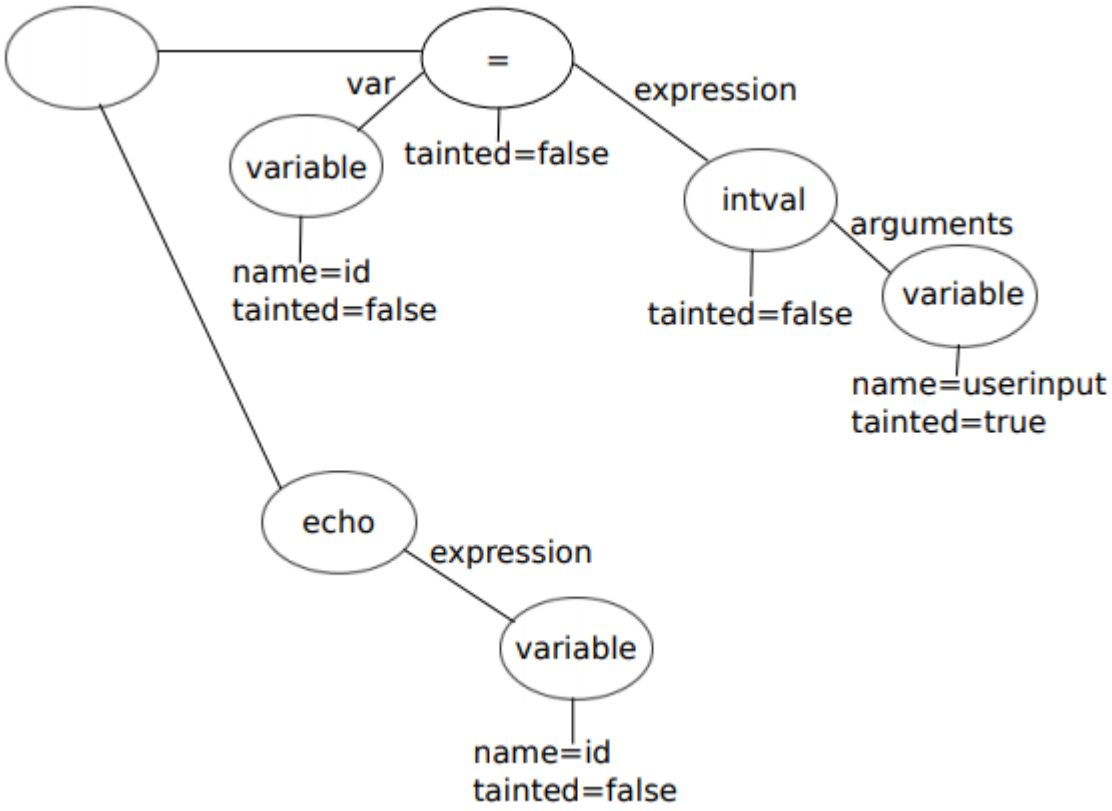

Рисунок 3.11 - Абстрактне синтаксичне дерево для рисунку 3.10

Область початкових змінних створюється на основі конфігураційного файлу. Призначення області – відслідковувати котрі змінні містять сліди вводу

користувача(taint). Потім відбувається прохід по абстрактному дереву: кожен вузол обходиться тричі. Перший крок оновлює область, коли стикається з вузлом, який змінює цю область. Якщо присвоюється змінна що має недовірені данні, то назначається мітка taint, і цей taint поширюватиметься на нову змінну. Наступний крок присвоює кожному вузлу атрибут taint. Далі, перевіряється чи у нас присвоюється змінна, чи викликається функція, якщо викликається функція, то ми пишемо sanitize функцію, що перевіре, чи в результаті її виконання залишиться taint мітка. Якщо ні, То на третьому кроці ми видаляємо цю фільтруючу функцію. На рисунку 3.10 показано код для абстрактного синтаксичного дерева рисунку 3.12, після видалення фільтруючої функції **intval**.

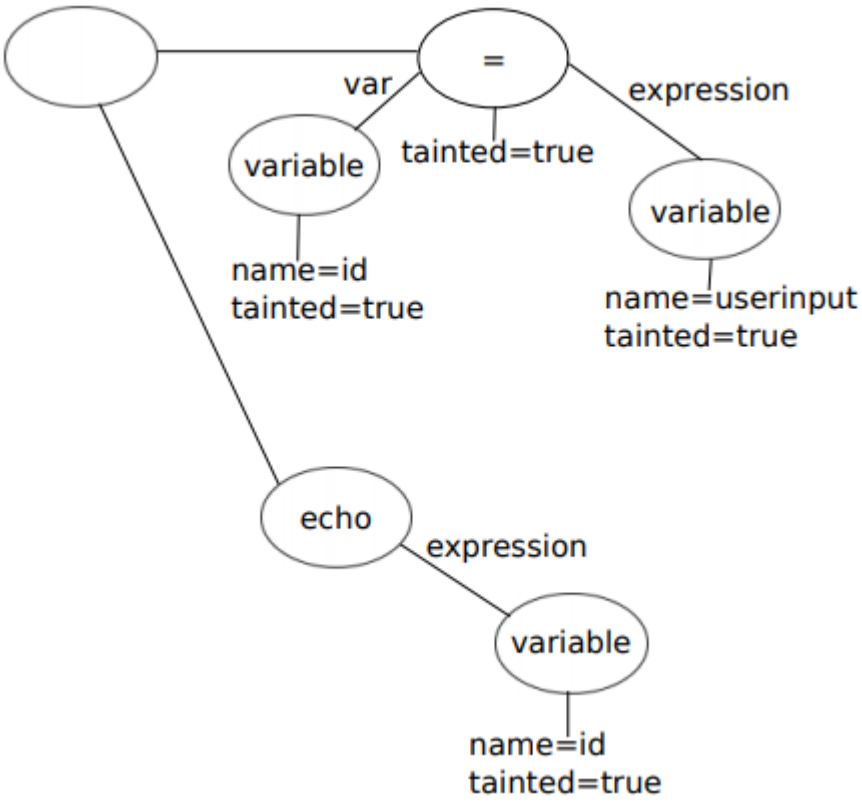

Рисунок 3.12 - Абстрактне синтаксичне дерево для рисунку 3.13

```
$id = $userinput;
echo $id;
```

Рисунок 3.13 - Код до абстрактного синтаксичного древа рисунку 3.12 після видалення функції фільтрації **intval**.

3.4 Метод статичного аналізу на чистоту(Static Taint Analysis)

Аналіз на чистоту - це аналіз того, як недовірені(або брудні - taint) дані обробляються програмою. *Недовіреними(taint) даними* називаються дані, що приходять від користувача через поля вводу, читання завантаженого користувачем файлу, чи іншими шляхами. Протилежні їй – *довірені*(trusted). Іншими словами, *властивість taint* значить що данні прийшли з недовіреного ресурсу. В контексті безпеки, аналіз на чистоту використовується для відслідковування *чутоливої* чи *недовіреної інформації.* Важливо проаналізувати, чи можуть недовірені дані досягти выходу в программі, тобто *поглиначів інформації*. Поглиначами інформації називаються зазвичай місце виводу даних користувачу, чи шляху виконання [14]. У разі якщо недовірена інформація може досягти таких виходів, то можна говорить про можливість вставлення уразливості в тій частині коду. Як приклад, користувач може заставити застосунок виводити довільні чи незаконні дані іншому користувачу, чи якщо недовірена інформація розглядається як чутлива, то вона може бути втрачена. Інформація може втратити taint властивість, якщо вона проходить через sanitation функцію(функцію очистки), у такому разі вона стає *довіреною* [32]. Після видалення цього флагу, дані розглядаються безпечними і виведенні *поглиначами інформації*.

```
$number = $userinput;
echo "Your number is: ", $number;
```

Рисунок 3.14 - Приклад як розповсюджується флаг taint. Вихідна змінна **$number** має цей флаг.

При виконанні методу Статистичного аналізу на чистоту, описаний вище процес виконується при перегляді вихідного коду веб-сайту чи застосунку. На рисунку 3.14 показаний приклад вихідного коду. Змінна **$userinput** є прямим вивідом від користувач і таким чином являється недовіреною і поміченою як taint. Виконання статистичного аналізу на чистоту на програмному коді покаже чи має місце впровадження уразливості в тому місці. Змінна **$number** присвоюється значеню змінної **$userinput**, котра помічена флагом taint. На наступный строчці кода та ж сама змінна виводиться користувачу(змінна досягає *поглинача інформації*). Через відсутність фільтрації змінної, на ній все ще флаг taint при досягненні неї виходу output. Аналіз закінчено, і він явно вказує на враливість.

```
$number = intval($userinput);
echo "Your number is: ", $number;
```

Рисунок 3.15 - Приклад того, як видаляється поширення флагу taint

Вихідна змінна **$number** уже не матиме флаг.

У прикладі рисунку 3.15 показаний майже такий же вихідний код як і в попередньому прикладі. Однак, у цьому варіанті змінна **$userinput** передається через функцію **intval** (повертає ціле число від поля input [20]) перед подальшим присвоєнням до **$number. Intval –** є sanitation функцією, тобто тою що очищує дані та робить їх *довіреними.* Коли змінна **$number** досягне місця виходу, то не матиме флагу taint і аналіз не покаже вразливості через sanitation функцію, що зробила дані чистими.

Метод статичного аналізу на чистоту має деякі недоліки, від яких страждають майже усі процеси статичного аналізу. Інколи може бути неможливим дізнатися всі можливі стани застосунку. Давайте розглянемо наступний код:

```
$key = rand();
$var = $array[$key];
```

У зв’язку з тим, що **$key** може приймати довільне значення, аналіз не може педбачити яке саме значення це буде.

Описані вразливості у даній роботі намагаються вивсти користувачу довільну інформацію. Таким чином, тільки поглиначі інформації, до яких зберігається флаг taint важливі, і тип розповсюдження має бути *прямим розповсюдженням* оскільки VulnerableCodeGen важливо що taint мітка зберігається аж до самого виводу данних, тобто їх потрапляння в поглиначі інформації, не вглиблюючись в те, як саме це відбувається. А *пряме розповсюдження* - коли нові данні запозичені з недовірених, тобто taint як в попередньому прикладі [22] де флаг taint від вводу користувача був напряму розповсюджений до змінної **$number.** Вразливість очікую данні у вигляді текстової строчки, бо усі операції що обробляють строки, наприклад цілочисельний привід данних зі строки, видалить флаг.

3.5 CSRF ін’єкція

До того, як VulnerableCodeGen може ввести CSRF-уразливість, вихідний код має бути попередньо підготовлено шляхом визначення, де реалізуються контрзаходи CSRF. Цеважко визначити статично, якщо дія захищена контрзаходами CSRF, оскільки існує кілька можливих контрзаходів. Одна з методик захисту від атак CSRF полягає у використанні випадково згенерованих токенів[6] [17] [5]. Не існує єдиного способу реалізації контрзаходів. Тому важко розробити алгоритм, щоб правильно визначити кожну частину контрзаходу від атаки, а потім визначити, як усунути їх для введення вразливості, і ще й різні варіації вразливості. Щоб зробити можливим для VulnerableCodeGen вводити вразливість, має бути вказана класифікація різних розділів контрзаходів (показано в 1.2) у вихідному коді. Можна вказати наступні розділи **Guard, Generate** та **Include**. Розділ **Guard** – сервер перевіряє, що запит надіслано навмисно, і якщо ні, припиняє виконання дії. У розділі **Generate**

токени CSRF генеруються та зберігаються, наприклад, у кукі-клієнтах. У розділі **Include**, токен CSRF включено у відповідь сервера на клієнті таким чином, що клієнт надасть токен в запиті на виконання дії автентифікації. Через те що код у розділі буде замінено або вилучено повністю, важливо, щоб розділи містили лише код, який пов'язаний з CSRF, а не з будь-якою іншою функціональністю, для запобігання руйнуванню веб-застосунку під час введення вразливості.

3.4.1 Приклад ін’єкції вразливості CSRF

На рисунку 3.16, сторінка захищена від CSRF за допомогою *Double Submit Cookies*, шаблону, що включає випадковий токен в выдправлену форму і cookie. Потім токени перевіряються на сервері [6]. Код вже був підготовлений для використання у VulnerableCodeGen. У конфігураційному об'єкті для кожного блоку коду визначається, що блок є частиною вразливості з ідентифікатор csrf changePassword. Вказується також, до якого розділу кожен блок належить.

Якщо VulnerableCodeGen запускався з конфігурацією рисунку 3.4, то вводив би CSRF уразливість з категорією **None**. Результат можна побачити на рисунку 3.17. Створена сторінка матиме відсутність будь-які сенсу для підтвердження того, що запит від користувача був навмисним. VulnerableCodeGen виконує різні зміни в різних розділах в залежності від категорії введеної вразливості. Якщо категорія втроєної CSRF - **None**, всі розділи видаляються, бо коли сервер отримує запит на виконання аутентифікаціїдії, валідація буде видалена, і сервер продовжить виконувати дію. Коли категорією є тільки запит **POST**, код урозділі **Guard** буде замінено кодом, який підтверджує, що запит є POST запитом. Інші розділи видаляються. У VulnerableCodeGen немає ні якої можливості втроїти CSRF з категорією **Multiple step**, тому що це вимагатиме розбиття початкової дії на кілька кроків. Потім, там має бути ін'єкція коду, який може відстежувати, на якому кроці знаходиться користувач. Наступний крок, необхідно ввести більше коду, щоб

дати користувачеві можливість отримувати інструкції для подальших дій. В **Guard** також буде **Referer header** заголовок з механізмом перевірки, який підтверджує, що заголовок містить значення, яке приймається до того, як сервер виконує дію аутентицікації. Коли VulnerableCodeGen встроює CSRF вразливість з категорією **Computable token**, розділ **Generate** замінюється на код, який генерує хеш md5 поточної дати і часу на сервері(При перегляді значення може здатися, що це випадкове значення, але його легко обчислити, знаючи data і time параметри сервера). Це значення тоді зберыгаэться в кукі-клієнтах, як у наведеному вище прикладі. Розділ **Include** замінюється так, що згенерований хеш md5 входить у розміщену форму. Нарешті, розділ **Guard** замінюється кодом, що перевіряє що cookie та відправлене значення співпадають.

```
<?php
if (isset($_POST["newpassword"])) {
/* VulnerableCodeGen
{ "identifier": "csrf_changePassword", "action": "guard"
} */
if ($_POST["csrf_token"] !== $_COOKIE["csrf_token"]) {
die("csrf attempt detected");
}
/*/ VulnerableCodeGen */
$newpassword = $_POST["newpassword"];
$user = get_current_user();
update_user_password($user, $_POST["newpassword"]);
}
/* VulnerableCodeGen
{ "identifier": "csrf_changePassword", "action": "generate"
} */
$csrf_token = base64_encode(openssl_random_pseudo_bytes(32));
setcookie("csrf_token", $csrf_token);
/*/ VulnerableCodeGen */
?>
<form method="post">
New password: <input type="password" name="newpassword">
<br>
<?php
/* VulnerableCodeGen
{ "identifier": "csrf_changePassword", "action": "include"
}*/
echo "<input type='hidden' name='csrf_token'"; echo " value='" . $csrf_token. "'>";
/*/Phulner*/
?>
<input type="submit" value="Change password">
</form>
```

Рисунок 3.16 - Веб сторінка захищена від CSRF

```php
<?php
if (isset($_POST["newpassword"])) {
    $newpassword = $_POST["newpassword"];
    $user = get_current_user();

    update_user_password($user, $_POST["newpassword"]);
}
?>
<form method="post">
    New password: <input type="password" name="newpassword">
    <br>
<?php
?>
    <input type="submit" value="Change password">
</form>
```

Рисунок 3.17 - Сторінка більше не захищена від CSRF

Висновки до розділу 3

У цьому розділі було розглянено як розроблялася модель та принцип роботи інструменту VulnerableCodeGen. Було детально обговорено як підготовлювати проект та використовувати інструмент. Не менш важливим було розглянуто Static Taint Analysis та парсинг коду у абстрактне синтаксичне дерево, та пояснено ціль використання цих методів.

4 ОЦІНКА ОТРИМАНИХ РЕЗУЛЬТАТІВ ВРОВАДЖЕННЯ

У цьому розділі буде показано результати роботи розробленої програмки VulnerableCodeGen а також оцінення згідно, критерій, встановлених мною у розділі 4. Було підготовлено два веб-застосунки: Wordpress та phpBB форум з подальшою їх оцінкою за допомогою веб-сканерів. Щоб оцінити час, необхідний на підготування проекту перед використанням програмки, було поведено тест з заміром часу.

4.1 Результати сканування впроваджених вразливостей

Існує близько 680 різних варіацій XSS уразливостей, однак лише декілька були реалізовані. Схожа ситуація полягає з CSRF.

Популярна на сьогодні CMS Wordpress [44] була підготовлена з XSS уразливостями у таблиці 4.1 і CSRF у таблиці 4.2. Веб-форму phpBB був підготовлений з XSS уразливостями в таблиці 4.3.

Таблиця 4.1 - XSS вразливості, що були встроєні у Wordpress

Вхід	Вихід	Тип фільтрації	Мутація
GET	Non JavaScript attribute - double quotes	None	Hi
GET	Non JavaScript attribute - double quotes	Blacklist	Hi
Stored	Normal tag	None	Hi
Stored	Normal tag	Blacklist	Hi
Stored	URL attribute - single quoted	None	Hi
Stored	URL attribute - single quoted	Blacklist	Hi
Stored	URL attribute - single quoted	Whitelist	Hi
Stored	URL attribute - single quoted	Encoding	Hi

Таблиця 4.2 - CSRF вразливості, що встроєні у Wordpress

Тип
None
Only POST request
Computable token

Таблиця 4.3 - Вразливості встроєні у phpBB

Тип входу	Вихід	Тип фільтрації	Мутація
POST	Normal tag	None	Hi
POST	Non JavaScript attribute - double quotes	None	Hi
POST	Non JavaScript attribute - double quotes	Blacklist	Hi

4.2 Можливість введення нетривіальних уразливостей

Із розділу 2 про аналіз сканерів веб-застосунків було вибрано **w3af** [39] та **Wapiti** [42] для проведення тестування результататів впровадження уразливостей програмкою VulnerableCodeGen.

Сканери були скачані із відкритих ресурсів та використовували виключно налаштування за замовчуванням. Спершу було перевірено згенеровані програмкою уразливості вручну, щоб точно знати що вони там є.

Результати сканування обох сканерів уразливостей на різних уразливостях можна побачити у таблицях 4.4 та 4.5.

Таблиця 4.4 - Результати сканування згенерованих веб-застосунків на XSS уразливість за допмогою w3af

Вхід	Вихід	Тип фільтрації	Мутація	w3af
GET	Non JavaScript attribute - double quotes	None	Ні	Так
GET	Non JavaScript attribute - double quotes	Blacklist	Ні	Так
POST	Normal tag	None	Ні	Ні
Stored	Normal tag	None	Ні	Ні
Stored	Normal tag	Blacklist	Ні	Ні
Stored	URL attribute - single quoted	None	Ні	Ні
Stored	URL attribute - single quoted	Blacklist	Ні	Ні
Stored	URL attribute - single quoted	Whitelist	Ні	Ні
Stored	URL attribute - single quoted	Encoding	Ні	Ні

Таблиця 4.5 - Результати сканування згенерованих веб-застосунків на XSS уразливість за допомогою Wapiti

Вхід	Вихід	Тип фільтрації	Мутація	Wapiti
GET	Non JavaScript attribute - double quotes	None	Ні	Так
GET	Non JavaScript attribute - double quotes	Blacklist	Ні	Ні
POST	Normal tag	None	Ні	Ні
Stored	Normal tag	None	Ні	Ні
Stored	Normal tag	Blacklist	Ні	Ні
Stored	URL attribute - single quoted	None	Ні	Ні
Stored	URL attribute - single quoted	Blacklist	Ні	Ні
Stored	URL attribute - single quoted	Whitelist	Ні	Ні
Stored	URL attribute - single quoted	Encoding	Ні	Ні

В першу чергу, вони знайшли тривіальні вразливості де вхідні данні були від GET параметра і напряму виводилися на сторінку без фільтрації. Коли були застосовано фільтр blacklist на поле вводу input, тільки один зі сканерів зміг знайти уразливість. Як і обговорювалося у розділі 3.1, сканерам важко знайти вразливість, коли дані попеедньо збережені перед виводом на сторінку. Це можна побачити у результатах тестів, так як жодний із сканерів не знайшов уразливості де вхідні значеня були спершу збеежені, а потім пізніе виведені.

Інша причина чому уразливості було важко виявити, що деякі форми повинні бути заповнені з прийнятними значенями перд тим як веб-застосунок навіть почне обробляти відправлені дані. Наприклад, якщо є форма відпвіді на пост на форумі і відповідний текст вразливий до XSS атаки. Якщо форма має поле, де необхідно ввести мейл адресу, то сканер має надати цей валідний мейл,

щоб сервер прийняв цю форму. Через те, що сканери ніяк не настроювалися а мають налаштування за замовчуванням, то не завжди можуть зрозуміти які дані треба подати у правильній формі, щоб пройти валідацію, тому і не знайдуть у цих лях уразлвостей.

Інша проблема полягає у тому, що веб-застосунки можуть бути дуже великими і містити безліч сторінок і сканер має пройти їх усіх, щоб перевірити на наявність зловмисної поведінки. Ця дія, перевірити усі сторінки може бути заважкою для сканера. Тому, деякі зі сторінок вони можуть просто пропустити, і пропустити вразливості що там можуть ховатися.

Тільки w3af мав підтримку сканувати CSRF уразливість і результати можна побачити у таблиці 4.6

Таблиця 4.6 - Результати сканування згенерованих веб-застосунків на CSRF уразливість за допомогою w3af

Тип	w3af
None	Так
Only POST request	Так
Computable token	Ні

Хоча w3af і знайшов уразливості у 2х із 3х категоріях, він також повернув багто непавдопдібних результатів. w3af доповідав про знайдені вразливості майже для кожній сторінці, де був включений чи POST чи GET параметр, тільки якщо один з параметрів не мав імені що виглядало як CSRF токен, чи значення парамеру. Через те що категорія **Computable Token** додає заховане поле input із назвою **volnurCodeGen_csrf_token** зі значенням md5, w3af прийняв це за CSRF токен і не розглянув це як уразливість. Цей результат доказує те що було обговорено у 2.1. Для сканера важко знайти CSRF без багатьох не вірних, не достовірних результатів . Людина все ще має визнати,

що уразливості існують. Результати тільки сумують те, де CSRF уразливості можуть бути присутніми.

4.3 Час підготовки проекту

Для оцінки часу, що треба для підготовки проекту людиною, що не працювала раніше із автоматизованим перетворювачем коду, я провів тест на студенті, з мінімальними навичками у програмуванні. Студенту було запропоновано підготувати новий проект з уразливостями.

Спершу, йому було надано коротку інформаці. про інструмент автоматизованого перетворювача коду VolnurableGenCode та показано приклади. Потім, студент отримав веб-застосунок та інструкції сторінок, у яких маються бути введені уразливості. Тестуючий не був знайомий із кодом програми до цього. Близько 10-15 хвилин йому треба було, щоб знайти місце де може бути уразливість та вставити необхідні ключові слова для роботи петворювача коду. Більше всього часу було потрачено експерементуючим на знаходження місця в коді, де можна вставити уразливість та розбір коду застосунку.

Даний тест показує що без будь-якого досвіду, треба дуже мало часу на те щоб розібратися як все працює і як користуватися автоматизованим петворювачем коду.

4.4 Подальші удосконалення

Усі застосунки можу та мають удосконалюватися, щоб розширювати свій функціонал та можливості. Деякі удосконалення що можна зробити у майбутньому:

- Зробити підтримку інших вразливостей, наприклад SQL ін’єкцій [11]

- Зробити sanitation функції для більшої кількості вбудованих PHP функцій. Зараз є лише деякі найнеобхідніші
- Коли вставляється XSS уразливість, виконується аналіз на чистоту. Зараз він може лише перевіряти з міткою taint массиви та змінні. Можна додати підтримку об’єктів, щоб можна було обробляти більше вихідного коду
- Було б цікаво перевірити код на платних сканерах, як вони показали б себе, а також найняти професійного пен-тестера щоб перевірити чи знайде він усі вразливості.

Висновки до розділу 4

У цьому розділі було розглянуто та оцінено результати виконання інструменту автоматичного перетворювача коду VulnerableCodeGen. Також було детально описано результати сканування підготовленого коду сканерами w3af та Wapiti. Оцінено і час, затрачеий на підготовку уразливостей звичайним студентом інституту та подальші способи удосконалення виконаної роботи.

ВИСНОВКИ

В результаті виконаної роботи, було детально розглянено XSS та CSRF атаки, їх функціонал та принцип роботи із прикладами, аналіз компаній, що займаються розробкою та захистом програмного забезпечення. Також, було проведено аналіз сканерів веб-застосунків, описано їх етапи роботи та перевірки уразливостей застосунків, основні недоліки та висвітлено список сканерів веб-застосунків на ринку, їх властивості, в результаті чого, було вибрано w3af та Wapiti для проведення експерименту із розробленим автоматизованим перетворювачем коду.

Розроблено та реалізовано інструмент автоматичного перетворювача коду VulnerableCodeGen, що здатен вводи уразливості у застосунки. Детально описано як підготовлювати проект та використовувати інструмент.

У результаті зробленого автоматичного перетворювача коду, було проведено експерименти по введеню уразливостей у веб-застосунки з подальшим їх скануванням сканерами Wapiti та w3af. Результати підтвердили те, що введені уразливості складно знаходяться сканерами, що розміщені у вільному доступі. Мало уразливостей було знайдено. Випливаючи з цього, можна зробити висновки, що автоматичний перетворювач коду був розроблений, як і планувалося, та повністю функціонує за сценарієм. Був оцінений і час, затрачеий на підготовку коду для використання інструменту тестуючим, без досвіду користування ним, та подальші способи удоскоалння виконаної роботи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 2011 Top 25 Most Dangerous Software Errors. CWE/SANS. [Електронний ресурс] - Режим доступу: https://www.sans.org/top25-software-errors - 06.05.2019

1 WhiteHat Security's Approach to Detecting Cross-Site Request Forgery (CSRF). Apr. 2011. [Електронний ресурс] - Режим доступу: https://www.whitehatsec.com/ - 06.05.2019

2 Security in Depth: New Security Features [Електронний ресурс] - Режим доступу: https://blog.chromium.org/2010/01/security-in-depth-new-security-features.html - 06.05.2019

3 Тестування методом Чорного ящика. [Електронний ресурс] - Режим доступу: https://www.anti-malware.ru/practice/solutions/web-applications-internal-threats-security#part - 06.05.2019

4 Boyan Chen [Електронний ресурс] - Режим доступу: https://www.researchgate.net/publication/220875877_A_Study_of_the_Effectiveness_of_CSRF_Guard. - 06.05.2019

5 Cross-Site Request Forgery (CSRF) [Електронний ресурс] - Режим доступу: https://ru.wikipedia.org/wiki/ Межсайтовая_подделка_запроса

6 Security from CSRF. [Електронний ресурс] - Режим доступу: https://www.owasp.org/index.php/CrossSite_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet - 06.05.2019

7 "Why Johnny Can't
Pentest: An Analysis of Black-Box Web Vulnerability Scanners". [Текст] / Адам Дюпе, Марко Кова, та Джовані Вігна - Springer, 2010, pp. 111– 131c.

8 mXSS Secured WebApplications by using innerHTML Mutations [Електронний ресурс] - Режим доступу: https://security.stackexchange.com/questions/46836/what-is-mutation-xss-mxss - 06.05.2019

9 Hypertext Transfer Protocol – HTTP [Електронний ресурс] - Режим доступу: https://ru.wikipedia.org/wiki/HTTPS - 06.05.2019

10 White Hat Security testing. Improper Input Handling. [Електронний ресурс] – Режим доступа: https://www.whitehatsec.com/glossary/content/improper-input-handling - 06.05.2019

11 Internet Security Threat Report [Електронний ресурс] - Режим доступу: https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf - 06.05.2019

12 Automated Mechanism for Secure Input Handling [Електронний ресурс] - Режим доступу: https://www.researchgate.net/publication/42803679_An_Automated_Mechanism_for_Secure_Input_Handling - 06.05.2019

13 OWASP: Vulnerability Scanning Tools. [Електронний ресурс] - Режим доступу: https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools - 06.05.2019

14 Отправка данных формы. [Електронний ресурс] - Режим доступу: https://developer.mozilla.org/ru/docs/Learn/HTML/Forms/Отправка_и_Получение_данных_формы - 06.05.2019

15 L.K. Shar and H.B.K. Tan. “Auditing the XSS defence features implemented in web application programs”. [Електронний ресурс] - Режим доступу: https://digital-library.theiet.org/content/journals/10.1049/iet-sen.2011.0084. - 06.05.2019

16 “Cross Site Request Forgery: A common web application weakness”. In: Communication Software and Networks (ICCSN), [Текст] / Mohd. Shadab Siddiqui and Deepanker Verma. 2011 IEEE 3rd International Conference on. IEEE, 2011.

17 The Web Application Security Consortium: Web Application Security

Scanner List. W3Techs. Usage of server-side programming languages for websites. [Електронний ресурс] - Режим доступу: https://blog.hackmetrix.com/wordpress-vulnerability-scanner/ - 06.05.2019

18 Web Security testing with Free Web Scanners [Електронний ресурс] - Режим доступу: https://www.anti-malware.ru/reviews/free_scanners_security_websites - 06.05.2019

ДОДАТКИ

ДОДАТОК А

Основні модулі коду застосунку

VulnerableCodeGen

CSRF Embeder

```
<?php
namespace VulnerableCodeGen\Embeder;

use VulnerableCodeGen\EmbederAbstract;

class Csrf extends EmbederAbstract {
 public function construct($action) {
$method = "_action_" . $action; //construct action
 $this->_method = $method;
 }

 public function embed ($code, $options) {
$res =  "// VulnerableCodeGen embedion start\n";
 $res = $res . "/* Old code:\n";
 $res = $res . $code . "*/\n";
$res = $res . $this->{$this->_method}($code, $options);
$res = $res . "// VulnerableCodeGen embedion end\n";
 return  $res;
 }

 private function action_generate ($code, $options) {
 if ($options->type === "COMPUTABLE") {
 $res =
<<<'CODE'
$VulnerableCodeGen_csrf_token = md5(date("Y-m-d"));
```

```
CODE;
  return $res;
    }
  return "";
    }


private function action_guard ($code, $options) {
  if ($options->type === "NONE") {
    return "";
    }
  if ($options->type === "ONLY_POST") {
      $res =
<<<'CODE'
if ($_SERVER["REQUEST_METHOD"] !== "POST") {
    die();
    }
CODE;
    return $res;
      }
  if ($options->type === "COMPUTABLE") {
      $res =
<<<'CODE'
if (!isset($_POST["VulnerableCodeGen_csrf_token"]) ||
$_POST["VulnerableCodeGen_csrf_token"] !== md5(date("Y-m-d"))) {
    die();
    }
CODE;
    return $res;
```

```
        }
    }

    private function action_include ($code, $options) {
        if ($options->type === "COMPUTABLE") {
            $res =
<<<'CODE'
echo "<input type='hidden' name='VulnerableCodeGen_csrf_token' value='",
$VulnerableCodeGen_csrf_token, "'>";
CODE;
            return $res;
        }
    }

    private $method;
}

?>
```

XSS Embeder

```
<?php
namespace VulnerableCodeGen\Injector;

use PhpParser\Parser;
use PhpParser\NodeTraverser;

use VulnerableCodeGen\EmbederAbstract;
use VulnerableCodeGen\NodeVisitor\taintChecker;
use VulnerableCodeGen\NodeVisitor\toReplace;
use VulnerableCodeGen\NodeVisitor\Scoper;
```

```
use VulnerableCodeGen\NodeVisitor\Scope;
use VulnerableCodeGen\PhpParser\Lexer;
use VulnerableCodeGen\PhpParser\NodeDumper;

class Xss extends InjectorAbstract {
const DEFAULT_function = "tainting";

public function __construct ($partConfig, $globalConfig) {
$this->_function = self::DEFAULT_function;

$this->globalConfig = $globalConfig;
$this->_partialConfig = $partialConfig;
}


public function inject($code, $options) {
$injectfunction = $this->_function;
$injectfunction = $options->function;
}

$function = "generate" . $injectfunction;

if (function_exist($this, $function)) {
return $this->$function($code, $options);
}
}

public function execute_tainting ($code, $options) {
if (!in_array($options->sanitation, $this->_partConfig->sanitation)) {
```

```
$options->sanitation = $this->_partConfig->sanitation[0];
}

if (!isset($options->output) || !in_array($options->output, $this->globalConfig->output)) {
$options->output = $this->globalConfig->output[0];
}
if (!isset($options->input) || !in_array($options->input, $this->globalConfig->input)) {
$options->input = $this->globalConfig->input[0];
}

$nodeDumper = new NodeDumper;

$parser = new Parser(new Lexer);

$statements = $parser->parse("<?php\n" . $code);

$traverser = new NodeTraverser;

$scoper = new Scoper($this->_initialScope, $options);
$traverser->visitor($scoper);
$taintChecker = new taintChecker($this->_sanitationFunctionsFactory, $options);
$traverser->visitor($taintChecker);
$toReplace = new toReplace($this->_sanitationFunctionsFactory, $options);
$traverser->visitor($toReplace);

$statements = $traverser->traverse($statements);
```

```
$res =        "// VulnerableCodeGen start (tainting)\n";
$res = $res . "/* Old code:\n";
$res = $res . $code . "*/\n";
$res = $res . "// VulnerableCodeGen end\n";
return  $res;
}

public function execute_removing ($code, $options) {
$res =      "// VulnerableCodeGen start (removeing)\n";
$res = $res . "/* Old code:\n";
$res = $res . $code . "*/\n";
$res = $res . "// VulnerableCodeGen end\n";
return $res;
}


public function setSanitationFunctions (Factory $factory) {
$this->sanitationFunctionsFactory = $factory;
}

public function setInitialScope (Scope $scope) {
$this->_initialScope = $scope;
}

public function setFunction ($function) {
$this->_function = $function;
}
}
```

```
?>

#!/usr/bin/php
<?php
require __DIR__ . "/vendor/autoload.php";

function prettifyString($text, $color = "0") {
  echo "\033[", $color, "m", $text, "\033[0m";
}

if($argc) {
  echo $argv[0] ." <config>\n";
  exit;
}


$configPath = $argv[1];
$config = json_decode(file_get_contents($configPath));

$injector = new VulnerableCodeGen\Generator();
$injector->grabFromCOnfig($config);

$injector->generate();

?>
```